Query Rewrite Optimization for NebulaStream

Daniel Dronov
TU Berlin
daniel.dronov@campus.tu-berlin.de

Tobias Engelbrecht
TU Berlin
tobias.engelbrecht@campus.tu-berlin.de

Riccardo Marin
TU Berlin
riccardo.marin@campus.tu-berlin.de

ABSTRACT

In database systems, the query optimizer has the potential to significantly enhance the query execution time by generating a quicker and more efficient query plan through the transformation of input queries. The crucial aspect of this component is the query rewriting phase, which employs various rules to modify, insert, delete, or rearrange operators within the query plan, resulting in an optimized and semantically identical plan. Such techniques are also applicable in stream processing systems. This paper focuses specifically on NebulaStream, an IoT data management platform. We implement rules that change the structure of queries in the NebulaStream system, before they get compiled and executed. By using these rewrite rules, the size of intermediate results during the query execution is reduced. In particular, our rules push down, reorder, split and merge filter operators and eliminate possible duplicates and redundancies. We conduct benchmarks with the application of the new rules. Our evaluation shows that - depending on the rule and the quality of the input query - the throughput of the system is increased and therefore NebulaStream can benefit from the rewrite optimizations.

PVLDB Reference Format:
Daniel Dronov, Tobias Engelbrecht, and Riccardo Marin. Query Rewrite Optimization for NebulaStream. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/nebulastream/nebulastream.git.

1 INTRODUCTION

With the advent of the Internet of Things and the ever-increasing need to process more data, traditional approaches based on batch processing systems are reaching their limits, and new solutions are needed. Continuous data generated from sensors and other devices needs to be analyzed in real-time since it cannot be entirely stored. Stream processing systems address this challenge. There are a variety of stream processing systems such as Apache Flink [7] and Spark [15] in the open source ecosystem or many proprietary technologies [11] [6]. They are able to consume continuous data streams, process them in a pipelined fashion and enable low-latency processing. However, due to its novelty, the field of stream processing is less developed than traditional data processing, with numerous areas still having potential for growth. One of the tasks is rewriting queries, which has the potential to greatly accelerate execution. For once the order in which the engine applies the different operators that are used to analyze the data can affect the amount of processing work. There may be numerous intermediary results that impede performance if they are not minimized, but the final outcome must remain unchanged irrespective of the operator execution order. In addition, some operators may be redundant, so there is a possibility of elimination. While in traditional database systems, this field has already been explored extensively, the optimizations entail increased complexity when applied to stream processing systems. One example is the novel stream processing system NebulaStream [16].

In this project, we aim to integrate a set of rewrite rules into NebulaStream. To accomplish this, we first familiarized ourselves with the internals of NebulaStream and grasped the existing optimizer's workflow. We then scrutinized a roster of suggested rules, filtering and prioritizing them accordingly. The rules have been tested and integrated into the NebulaStream platform. We conducted experiments to measure the performance of our implemented optimizations. Some rules show a negligible or even slightly negative impact on the throughput of the system, while others lead to a significant improvement.

The contributions of the project are the following:

(1) Analysis of a proposed list of rewrite rules. We decided whether each of the rules of the provided catalog could be applicable or not to our case. The selected rules have been ordered by priority, depending on the expected impact of the optimization rule.
(2) Implementation of a set of rewrite rules. We implemented the rules and integrated them into the query optimizer of NebulaStream.
(3) Performance evaluation of the impact of the new rewrite rules. We assess the throughput of the system among different queries and conditions.

2 RELATED WORK

While the topic of query optimization is well-researched, the combination of both rewrite optimization and streaming systems is barely covered by the scientific literature. A rewriting approach targeting aggregations in materialized views can be found [9]. The paper proposes a new operator called generalized projections, which is an extension of duplicate elimination. The generalized projections are used when computing aggregations and pushed down below selections. Not all the presented ideas may be used in streaming contexts; on the other hand, the main work, which is about reducing intermediate results by pushing down operators, strongly resembles our contribution. The work describing the cost-based query transformation performed in Oracle [5] is one of the few papers including an in-depth performance analysis of the impact

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

of the rules. Traditional database systems rely upon relation cardinalities when estimating the cost of the query plans. However, this information is often missing for streaming inputs, since the system is dealing with an unbounded input. The limitation of cardinality estimation is addressed by Viglas et al. [13]. The authors propose an optimization framework built upon a rate-based approach for cardinality estimation. A relevant opportunity for optimization in data streams comes from the window operator [14]. Window containment and window correlation techniques identify redundant computations and try to reuse already available results. Another line of work addresses window slicing, which divides the whole window into smaller chunks and calculates the aggregates by aggregating the sub-aggregates over the small chunks. Hirzel et al.[10] provide a broad summary of different optimizations in streaming applications. They give a theoretical overview of the performance benefits and limitations in the areas of operator reordering, redundancy elimination, operator separation, fusion, and fission, as well as areas unrelated to changing the operator structure in the query. However, this broad overview only leaves space for an exemplary view that gives an idea of which aspects need to be considered. Our project implements rules that differ slightly from those covered in this overview and take into account additional situations for which we consider both correctness and performance. It is worth mentioning the presence of open-source DBMSs that implement query rewrite, such as Hyrise [3], Noisepage [4], and DuckDB [2]. Although they do not handle streams, they are a good reference for evaluating the approaches used in the database systems landscape.

3 BACKGROUND

In this section, we provide some background information on which our contribution is based. We introduce stream processing systems (Section 3.1) in general and the specific case of NebulaStream (Section 3.2). Moreover, we describe query rewrite rules in general and how query plans are affected.

3.1 Stream processing

Stream processing describes techniques for handling the continuous arrival of data, and is often applied to large data sets. Unlike batch processing, which also handles large amounts of data, the focus is on the continuous arrival of data and often a need to process results as the data arrives, for which batch processing systems are not suited [12]. Stream processing systems are the ideal solution to minimize the latency, while batch processing systems are more optimized for throughput. Another key difference is that with streams, data items are pushed (rather than pulled) into the system, and a job graph of long-running operators continuously produces results. Conversely, with batch processing, the computation takes place in stages, with intermediate results materialized either in memory or on disk. In stream processing, numerous devices generate multiple streams, each comprising tuples of data that may be produced periodically or irregularly. Each tuple contains values from a well-defined schema, including a timestamp indicating when the tuple was generated by the device. The timestamp or other additional information about the production time of a tuple is essential for the stream processing engine to apply strategies for out-of-order or missing tuples, and the well-defined schema enables the engine to process the data values of the tuples. In stream processing, tuples can either be processed individually, in which case they can also be processed immediately, or multiple tuples can be concretized to be processed together. This is necessary to perform aggregations or joins, and stream processing systems use different types of windows that group tuples together according to their production time.

3.2 NebulaStream

NebulaStream is a stream processing engine designed for the Internet of Things (IoT). This means that it focuses on distributed parallel processing of many different streams. It is highly scalable and works among heterogeneous hardware platforms and workloads. NebulaStream follows the server/worker paradigm: a cluster is made up of a coordinator and one or more workers. The former runs on the cloud and coordinates the deployment, the latter consists of data sources, sinks, or intermediate nodes, contains the execution engine, and performs the actual data processing tasks.

All queries in NebulaStream process data from a specific logical stream. Streaming data is then consumed by operators. Those supported by the system are: filter, projection, map, rename, union, and window. The possible types of windows are tumbling, sliding, or threshold windows. Windows may compute aggregations or be joined with another operator of the same type. The records of the results are transferred with the sink operators, which can forward data to an external server, a file, or simply print the output.

3.3 Query Plans

The queries can analyze streams from different sources and support joins, aggregations, selections, and more. For joins and aggregations, we need the aforementioned window operator to work on a concrete set of data tuples.

A simplified workflow for executing a given query is the following. The user's input query is first parsed into an abstract syntax tree. This is then converted into the equivalent logical query plan. As a next step, the system generates a mapping of each logical algebra expression to the optimal equivalent physical algebra expression. The resulting physical query plan is then compiled and executed in a distributed fashion.

The query rewrite can be applied after obtaining the logical query plan (Figure 1). In this way, a new optimal logical query plan is generated. One of the key ideas behind query rewrite is reducing the size of the intermediate results passed along the plan. The logical query plan is a tree consisting of logical operators. As long as the semantic meaning of the query is preserved, the operators can be moved or changed in the tree. The syntax and options for these operators may differ between systems, but in general, each system provides common operators.

A filter operator selects tuples and forwards them to the next operator if and only if the tuple satisfies the conditions that are specified in the filter predicate. Furthermore, a projection operator only forwards the specified columns of each tuple. The join operator takes multiple input sources and outputs tuples matching a certain join condition on the columns from both inputs. Windows split the stream into buckets of defined size, over which we can apply computations, e.g. aggregations or joins. In general, windows

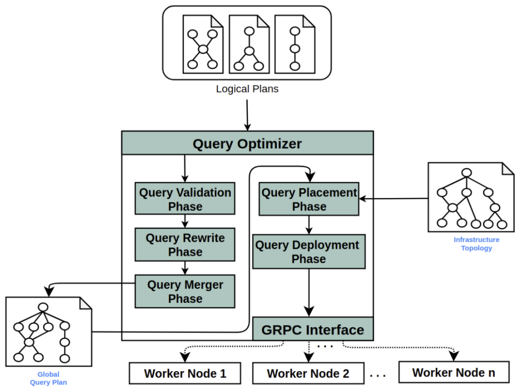


Figure 1: The role of the Query Optimizer in NebulaStream architecture

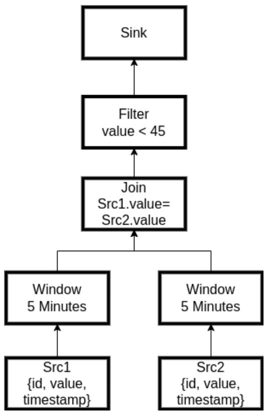

Sink
Filter value < 45
Join Src1.value= Src2.value
Window 5 Minutes
Window 5 Minutes
Src1 {id, value, timestamp}
Src2 {id, value, timestamp}

Figure 2: Example query plan

are defined by a window type, a window measure, and a window function.

An exemplary query plan valid for NebulaStream is depicted in Figure 2. Two sources - of which one selects a window of 5 minutes - are joined on the equality of the common attribute value, then the result is selected according to a filter on the same attribute and finally returned through a sink.

4 SOLUTION

We propose a solution consisting of the implementation of several rewrite rules that can lead to faster query processing. The potential optimizations range from eliminating redundant elements in the query to completely restructuring the query plan. For example, the query plan shown in Figure 2 can be improved. Moving the filter closer to the sources does not change the semantics of the query; at the same time, it reduces the number of tuples in the intermediate result passed to the join operator. Each rule is explained in detail and illustrated with examples in the following sections.

4.1 Query Rewrite

As described in section 3.3, the query rewrite step is called inside the optimizer component, which runs after the logical query plan generation and before producing the physical query plan. In the case of NebulaStream, the optimizer has 5 phases: query merger, query placement, query rewrite, query signatures, and query validation. In the query rewrite phase, the query plan is passed to a sequence of individual rules in a certain order. Finally, the transformed query plan is returned. The order of application of the single rules has to be carefully pondered; for example, if splitting an operator enables a subsequent push-down of the operators produced by the split, the split operation has to necessarily precede the push-down rule.

Another important constraint to consider is query compilation, which converts the query plans into their compiled version. NebulaStream employs query compilation to avoid the inherent overheads of interpreter-based engines (many function calls, field access offset, type checking) and achieve a degree of data locality close to an optimized handwritten C program. This is done by fusing operators together as much as possible. Pipeline-breaking operators force the system to materialize results, thus all operators of the same operator pipeline are compiled into a single operator.

4.2 Filter Push Down Rule

The filter push-down rule attempts to push filter operators as far down the query plan as possible. In this way, the filter operator reduces the number of tuples that each subsequent operator has to process, and it is therefore beneficial for the overall execution if the selection of tuples is done as early as possible. There are some limitations to the benefits, as we may not gain much if the filter operator selects almost all tuples, or if optimizations in other places, such as pipelining during execution, fuse the operators that have now changed the order together anyway. However, we can still reduce the overall workload if we have a somewhat selective filter that gets pushed below a pipeline breaker. Rough assumptions in relational databases estimate that filters with range attributes filter out 2/3 of the tuples and equal attributes filter out even more [8, Chapter 16.4.3]. Thus we can assume that filters in streaming systems rarely select nearly all attributes, and often reduce the workload for the following operators. While the system benefits most when the filter is pushed below a pipeline breaker, it is essential that the filter can be pushed under other operators, as this may prevent the filter from being pushed below a pipeline breaker. As we move down the filter, we can analyze the operator below in isolation and move the filter below that operator if possible, i.e. the end result of the query is still the same regardless of the data set. Therefore, we will take a deeper look at the individual operators in NebulaStream and what aspects need to be considered if we want to push a filter operator below them.

Map Operator. The map operator updates the values of one or more existing columns, or of a newly created column. To update the values it may also take values from other columns as well. Since we want to push the filter operator below this operator, we need to ensure that it still selects the same tuples. This is the case if any part of the predicate is neither one of the updated columns nor one of the newly created columns, in which case the query would even crash. It does not matter if the columns that are checked in the predicate are used by the map to calculate the values of an updated column, as the values of these columns themselves are not affected, and therefore the filter still selects the same values.

Projection Operator. The projection operator has some specified columns, and for each tuple that it takes as input it outputs the tuple with only the specified columns. It is always possible to push a filter operator below any projection operator. The only columns that can be accessed in a filter predicate above a projection operator are those columns that are the output of the projection operator none of which have changed values for any tuple and they are furthermore also contained in the input of the projection operator. In NebulaStream the projection operator has the specialty that it is the only place where columns can be renamed. The column values are still the same, but the name of the column could differ and this could affect the filter as well. In the case that a column that is part of the filter predicate gets renamed, we can still push the filter below the projection. However, we need to adjust the predicate accordingly. This means that all the columns in the predicate that access a renamed column have to update their internal column name to the name before the renaming of the projection. This information is easily available and allows us to push the filter with an adjusted predicate below this projection operator.

Window Operator. The window operator is used in streaming contexts to group tuples of a specified constraint - often a time interval - and this enables the developer to perform analysis on multiple tuples. This is needed for other operators like joins and aggregations, that can only work on concrete data tuples. However, the filter operator only requires individual tuples to determine which tuples to select and does not require the entire set of tuples within the window. It is possible to move the filter operator below the window operator, enabling the filter to select a tuple immediately after its production, followed by the window operator determining if the filtered tuple corresponds to a specific window.

Join Operator. The join operator is where we expect most performance benefits, as it is a pipeline breaker and it is quite difficult to push it through other pipeline breakers, such as aggregations. A join processes tuples from two input operators and outputs new tuples that contain the columns from both inputs. This is crucial for the filter operator since it can be potentially pushed down below. Nevertheless, we can only push it below the join in certain scenarios. The predicate of the filter might compare two columns from both of the joins input operators. In this case, it is not possible to push the filter below any side of the join, as we would not have enough information to perform the selection. In general, we only push filters below one side of the join if all columns that are accessed in the filters predicate come from this side of the join. They don't need to come from the same source, but they all need to be part of the output of one of the join child operators.
We identified one special situation in which we perform an additional optimization. In the case that the filter predicate only accesses those columns that are part of the join condition, we can push a filter to each side of the join. In NebulaStream only equi-joins exist at the moment and we consider filters that only access one column in the predicate, which is part of the join condition. Even though the filter selects tuples based on the newly created column after the join is done, the column it accesses is determined before any optimization step and belongs to one source. We can push the original filter to the side containing this source and at the same time we create a new filter operator that has a similar predicate with the only difference: all references to the column are updated to the column that makes up the other side of the join condition. This whole optimization reduces the number of tuples that the join has to consider. It is valid because if we filter tuples based on the join condition on one side of the join, we would not find a matching tuple for this value even if we have not filtered this value in the other branch.

Grouped aggregation. Aggregations, like joins, are pipeline breakers, so it could provide a bigger benefit if we manage to push filters below aggregations. However, this is rarely possible as the aggregation changes the values of the fields. There is only one situation in which we can be certain that the values of columns stay the same and this happens in the case of a grouped aggregation for those columns that are part of the group by operation. If we want to filter some values depending only on columns that are part of the group by operation it doesn't matter if we do that before or after the aggregation. For example, we could have a dataset for which we want to have the mean age grouped by countries, but we are only interested in certain countries. The result will be the same regardless if we filter out other countries before or after the aggregation. This also works if the filter is dependent on multiple columns that are all part of the group by clause. However, there is no optimization possibility as soon as the filter is dependent on at least one column that is not part of the group by as this column will have its value changed.

Operator schemas. In NebulaStream each operator also keeps track of the input and output schema at each point of execution. Namely, the input schema consists of the columns of tuples received by the operator. The output schema represents the columns that result from the tuple processing. It serves as the input schema for the subsequent operator. An operator may also receive input from two operators, in which case it has two input schemas which each correspond to one output schema of the previous operator. The filter operator is quite simple in this regard, as it does not change the schema of the tuples it receives. Thus, even if we remove a filter operator, the input and output schemas of the previous and next operators are still the same. However, the filter gets pushed below other operators and during this process, the input schema of the tuples it receives may change. To make the schemas consistent again, we set the input and output schema of the filter operator equal to the output schema of the operator that we push above. Since the filter operator does not affect the schema, we do not need to update the schema of any other operator.

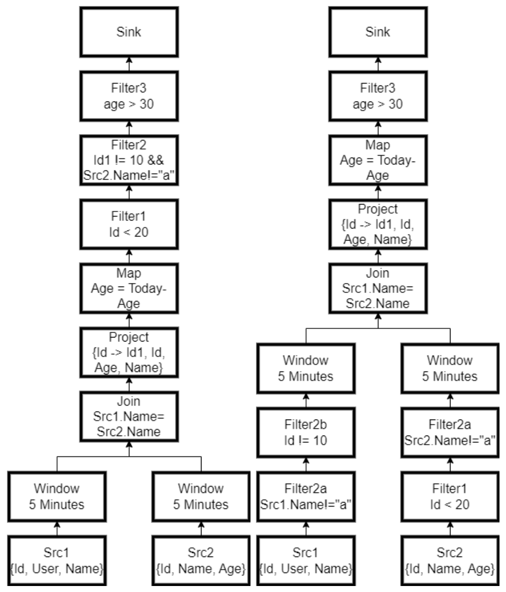
Multiple Query Fusion. Currently, we have only looked at query plans in isolation. This is a simplified view because modern data processing systems often handle many queries simultaneously and when processing them, they fuse multiple query plans into a unified plan that delivers multiple results. This is done to optimally exploit intermediary results that can be used by multiple queries and therefore prevent us from doing unnecessary recomputations. In NebulaStream this step is done before any of our optimizations take place, and can be easily seen in the query plan if any operator forwards his results to more than one parent operator. We need to consider this when we push down filter operators below any operator. Before checking any of the rules specified for each operator, an extra check is necessary. The operator through which the filter is intended to be pushed below must have only one parent. If multiple parents are present, it indicates that the sub-plan below this operator is utilized by multiple queries. Hence, the filter cannot be pushed into this sub-plan since it is solely used in the sub-plan above this operator. Inside any sub-plan we can still push the filter according to the rules we have specified for each operator, we just need to do this check at each point where we want to push the filter.

4.3 Filter Split-Up Rule

The filter split-up rule divides a single filter with a predicate consisting of conjunctions into independent multiple filter operators without conjunctions. It is a supplementary rule that does not provide any benefit on its own, but it enables an enhancement on top of the filter push-down rule. Since we are dependent on the columns that a filter accesses if there are fewer, we have a better chance of pushing the filter down. From the perspective of the original unsplit filter, we might be able to push parts of it down the query plan, or even the complete filter if we can push the parts to different child operators. Still, sometimes we might not be able to push down any part of this filter. Since other rules that are implemented in this project benefit from this, we apply them before any of the other optimizations that we present in this project.
The way this rule works is that it looks at predicates and the expressions within. The expressions are read according to the order they are analyzed. So we look at a top-level expression that might consist of multiple other expressions. If this top-level expression is a conjunction, we split this filter into two filters that each have a predicate with a new top-level expression that is either the left or the right side of the conjunction expression. This results in the same output, because we still select everything according to the first expression. Afterward, we select all the tuples from the set that match the second expression and also match the tuples already selected by the first expression. To further enhance this rule we apply De Morgan's rules to the predicates if possible. We can convert a negated disjunction expression into a conjunction expression by negating each part of the conjunction. Additionally, an expression that is negated twice can be rewritten as the original expression. Since the predicates might contain more conjunctions than just the top-level expression, we apply this rule recursively. After splitting up one conjunction we look at both of the resulting filters and apply this rule to both of them. After the recursion is finished, no filter predicate will have a conjunction as its top-level expression.

4.4 Example filter split up and filter push down

We will use figure 3 to explain the details of the filter push-down rule as well as the filter split up rule. As a first step we will describe the input query, to get an understanding of how the query transforms the data.
It receives data from two sources, with one source producing tuples of the schema `Id, Name, Age`, and the other with the schema `Id, User, Name`. After that, the data of both sources is joined together with the join condition that the names of both tuples have the same value. In order for the the join to be possible both data streams need to be concretized which is achieved with the window operators. After the join, we have a projection that gets input tuples with five columns `Name`, `Src1.Id`, `User`, `Src2.Id`, `Age` and produces output tuples with the schema `Name, Id1, User, Id, Age` where `Id1` was produced by `Src1` and `Id` by `Src2`. After the projection, we have a map operator that changes the value of the column `Age`. After these operators, we have three different filters that select only certain tuples. The first filter selects only tuples that have an `Id` that is bigger than 20. We know that this is the `Id` column from `Src2` since the other Id column was renamed in the projection to Id1. The second filter consists of a conjunction that selects based on `Id1` and `Src2.Name`. NebulaStream always assigns a source to each attribute so all of the other attributes have their corresponding source assigned as well, we have just decided to leave the names out of the figure for the sake of clarity. The third filter selects only tuples with an `Age` that is greater than 30.
We will use this exemplary plan to explain the optimizations that the filter split-up rule as well as the filter push-down rule provides and what constraints they have. The filter split-up rule is performed before the filter push-down rule. The only filter that is affected by this rule is filter two and we can see the result of this rule in the optimized query plan (in Figure 3) where we have filter two-a and two-b that each contain one side of the conjunction. In the optimized plan they are already pushed down the query plan which will now be explained for each individual filter as follows. First, we will push down filter one. Filter one only accesses the `Id` column which we know is from source two. It can be pushed through the



(a) Unoptimized input query (b) Query after applying filter split up and filter push down

Figure 3: Example Optimization

map without a problem since the map only changes the `Age` column. Filters can always be pushed through the projection operator and in this case, the accessed `Id` column is also not renamed. Since the filter only accesses one column all of the accessed columns belong to one branch of the join, which in this case is the right branch that contains source two. The next filter that is pushed down is filter two-a. It accesses the column `Src2.Name` and can be pushed through the map and projection in the same manner as filter one without any problems. At the next step, it is pushed through the join and could be pushed to the right branch that contains source two. The result would still be the same as all the tuples from source one with the filtered name would not find a corresponding tuple from source two. However, this enables us to do the special join optimizations, where we create a duplicate filter that filters tuples with the same value for `Name` which are produced by source one. So in the end we have two filters two-a that are each below one side of the join and access the column `Name` of the corresponding source. The next filter will be two-b. It accesses the column `Id1` which is the renamed column `Id` of source one. It can be pushed below the map in the same manner as the previous two filters, but pushing through the projection is more interesting. It can still be pushed through the projection, but the column it acts on needs to be renamed in the same step to its original name `Id` (in reality `Src1.Id`). Afterwards, it can be pushed to the left side of the join since all the columns that are accessed come from source one. All three filters can be pushed through the window as we don't have any constraints for this operator. The next filter is filter three, which accesses the column `Age`. Since the operator below it is now the map we can not push it at all since this map changes the value of the column `Age`. In this example, it is possible to push multiple filters below a join which could significantly improve the processing time for the join itself.

4.5 Filter Merge Rule

The filter merge rule aims at reducing the number of operators in the query plan, which can potentially accelerate the query processing time. It identifies sequences of consecutive filters. The filters are combined together into one filter where the predicate is a conjunction of the predicates of the original filters. It is especially useful after splitting up and pushing down the filters, which is also the case in our implementation.

4.6 Filter Reordering Rule

This rule could provide a benefit for the execution since the selectivity of each filter is different. Executing a filter that selects fewer tuples first, will leave fewer tuples to process for the subsequent filters to work on. In reality, the filters will not be executed individually, but - thanks to operator fusion in the compilation phase - in one pipeline that looks at all the tuples once. This does not mean that this optimization is unnecessary; we still need to provide an order in which the different conditions are to be checked before they get fused into one pipeline. The compiler does a similar optimization as it can reorder conditions within if-statements in order to check fewer conditions. However, what can be done by our rule is to provide an already optimal order for the compiler. In this way, we offload some work of the compiler, since it can already start with the optimal order and does not need an adjustment period in which it might find this optimal order as well. Actually, this rule is only partially implemented in the NebulaStream system. Determining the selectivity of a filter is not trivial when dealing with streaming data. It needs the system to keep track of a lot of statistics that were outside the scope of the project and are not implemented in the system yet. However, NebulaStream already provides a filter selectivity that can be manually set so that the rule can be tested and integrated as soon as the system calculates the selectivity for the filters.

4.7 Redundancy Elimination Rule

This rule is responsible for reducing redundancies present in the filter predicates. Namely, three strategies can applied:

(1) Constant moving and folding
Constant moving transfers all the constants to one side of the predicate. `filter(Attribute("id") + 10 > 10)` becomes `filter(Attribute("id") > 0)`, for instance. The folding operation collapses simple arithmetic expressions into its result. A potentially redundant query statement such as `filter(Attribute("id") > 5+5)` is therefore rewritten as `filter(Attribute("id") > 10)`.
(2) Arithmetic simplification
This rule applies to arithmetic expressions to which the answer is known. In particular, it detects simplification opportunities given by the properties of numbers 0 and 1. `filter(Attribute("id") > Attribute("value") * 0)` is, for example, reduced to `filter(Attribute("id") > 0)`
(3) Conjunction and disjunction simplification
`filter(Attribute("id") > 0 && TRUE)` is transformed into `filter(Attribute("id") > 0)`. This last strategy is at the moment not applicable, as boolean values in the queries are not supported.

The rule is not currently used in the system - since the compiler already does most of these optimizations automatically - but can still be applied and expanded in the future.

4.8 Duplicate Operator Elimination Rule

The objective of this query optimization rule is to enhance query performance and optimize resource usage by identifying and eliminating duplicate operators within a query plan. This rule specifically targets instances where the query plan contains multiple identical operators of the same type, with a focus on removing as many duplicates as possible. Notably, this rule is currently implemented for Filter and Projection operators. However, it operates with an awareness of certain special cases that require careful consideration:

(1) Handling Joins: In scenarios where there are duplicate projections and filters originating from two distinct branches of a join operation, the rule refrains from removing them. This approach is taken to avoid inadvertently increasing the generation of intermediate results, which could potentially degrade performance.
(2) Considering Maps: Duplicate filters that are subject to manipulation by a mapping operation in between them are not eliminated. This is because their removal could lead to

either an increase of intermediate results to be mapped or a modification in the query's output, both of which need to be avoided.

5 EVALUATION

In the early stage of the project, we considered each case and estimated which rules might be most beneficial. In the evaluation phase, we benchmarked them to verify our assumptions. More importantly, we could assess the impact of rewriting rules.

5.1 Methodology

The experiments have been run with the following setup. We measured the performance on a laptop running Ubuntu 22.04 with 16 GB main memory and processor i7-1165G7 @ 2.80GHz with 8 cores. For the measurements, we used the module `nes-benchmark`, which was already part of the NebulaStream project, version 0.5.18. It loads a *yaml* configuration file including the benchmark settings and generates a CSV file with the result. The parameters that can be tuned are query, number of threads, buffer size, and number of buffers to produce. We used the provided YSB (Yahoo Streaming Benchmark) data generator to ingest data into the system. We allocated a buffer size of 4096 bytes and generated a number of buffers in the order of a hundred thousand, depending on the query. We created two settings: a normal condition, where the number of processed tuples is stable, and one where the system cannot keep up with the number of ingested tuples and has to introduce a performance deterioration. We ran benchmarks 10 times and averaged the obtained result. Regarding the measured metrics, we focused on throughput and latency, which are the main performance indicators in data processing systems. The performance of each rule was measured both in isolation and in combination with other rules.

Queries. We have created 5 different queries [1], each of which has optimization opportunities. Q1 has consecutive filters on one source of data. Q2 has also only one pipeline, which consists of a sequence of several filters, maps, and projections. Q3 joins two sources, of which one uses a tumbling window. A filter is present close to the final sink and can be pushed down. Q4 has the same structure as Q3 but has more filters both close to the sink and to the source. Q5 involves a union of two streams and a filter that can be split up and pushed to both sides of the join.

5.2 Results

First of all, we estimated the footprint of our optimizations. We ran the unit tests calling our rewrite rules and measured the average execution time over 1000 runs. The additional time to restructure the query plans falls in the order of microseconds and ranges from 50 to 250 us when using 4 to 16 operators. This information can also be seen in Figure 4 and shows a linear trend. Therefore, we reckon that the solution is scalable and has a small overhead since the execution time linearly increases with the number of operators.

The execution of Q1 does not show significant differences between the baseline and the optimized versions (Figure 6). In fact, no optimization is applicable for this case. We can also observe that the overhead of the rules execution is restrained.

Q2 has some optimization potential given by the possibility of pushing down the filters through some intermediate operators.

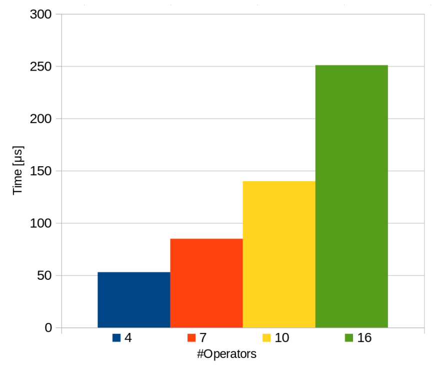


Figure 4: Average additional execution time

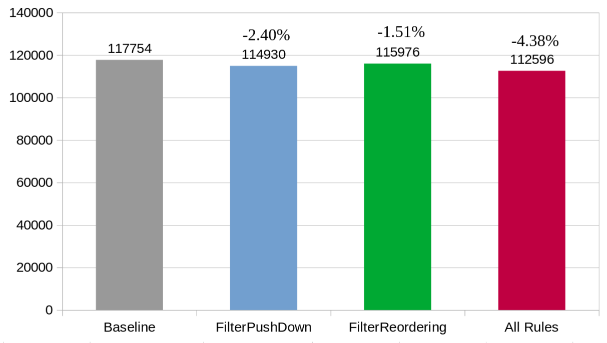


Figure 5: Q1, throughput comparison

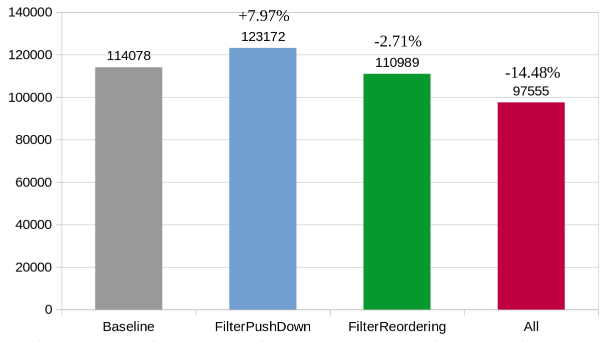


Figure 6: Q2, throughput comparison

The application of filter push-down rules is actually increasing the throughput by almost 8%. The other rules do not apparently bring any benefit to this query.

Q3 and Q4 demonstrate in particular the impact of the filter push-down rule, which duplicates the filter and pushes it down on both branches of the join (Figure 7 and 8).

We also measured Q4 under stress. In all cases a performance degradation is evident: the processed tuples per second dramatically decrease and a spike in the latencies is to be found. However, the increase in the latency is more gradual and slower when applying the filter push-down rule (Figure 9).

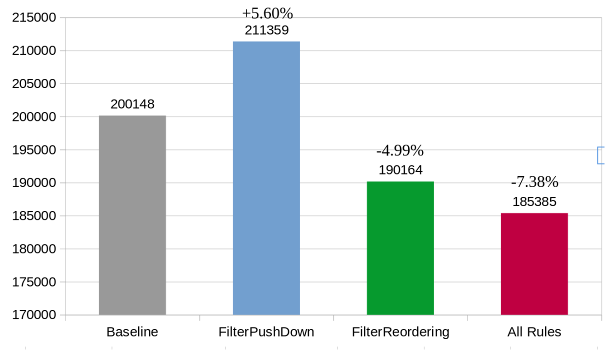


Figure 7: Q3, throughput comparison

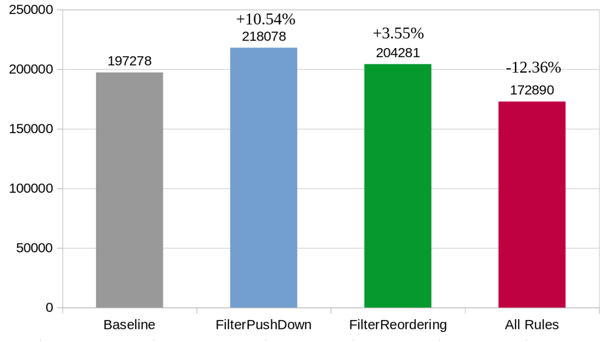


Figure 8: Q4, throughput comparison

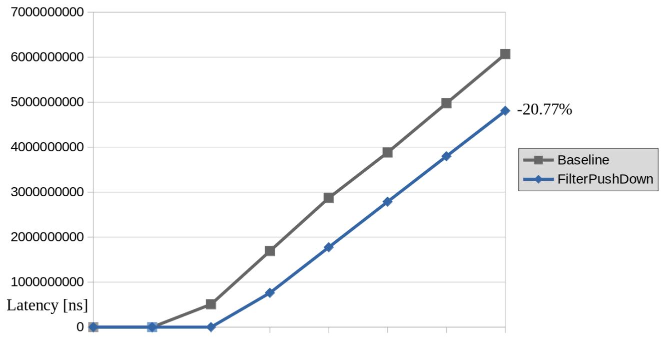


Figure 9: Q4 latency under stress

5.3 Summary

As expected, the filter reordering does not bring any concrete benefit at the moment. Since the selectivity estimation is not implemented yet, it only generates a small additional overhead. The application of the rules combined seems to slow down the system by a significant amount when no optimization is possible. This indicates that the rule filter split up and merge needs to be refactored in order to be more efficient. The filter push-down rules seem to be the most beneficial. It helps both improve the throughput in normal conditions and decrease the latency when the system is under stress.

Regarding future work, we identify the duplicate operator elimination rule as a potentially helpful rule. We assume it can be impactful on the execution time. It should detect redundant operators in the query plan - in particular filters and projections - and remove the unnecessary duplicates.

6 CONCLUSION

We investigated the potential of query rewrite rules in a less-explored setting, that is stream processing. The implemented rules, including filter split-up, filter push-down, filter merge, filter reordering, and redundancy elimination, had varying impacts on processing time. Although some rules had a slightly negative effect, they would benefit query computation as soon as integrated with new features (e.g. query selectivity). Most rules lead to a small improvement in throughput. In addition, these rules reduce the processing latency when the system is put under stress. This way we demonstrated that NebulaStream, a novel state-of-the-art system, can leverage some of these rules to improve the performance of the streaming queries.

REFERENCES

[1] [n.d.]. Benchmarks repository. https://github.com/rimarin/query-rewrite-optimization-nebulastream

[2] [n.d.]. DuckDB Rewrite Rules. https://github.com/duckdb/duckdb/tree/master/src/optimizer/rule

[3] [n.d.]. Hyrise Rewrite Rules. https://github.com/hyrise/hyrise/tree/master/src/lib/optimizer/strategy

[4] [n.d.]. Noisepage Rewrite Rules. https://github.com/cmu-db/noisepage/blob/master/src/optimizer/rules/rewrite_rules.cpp

[5] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zait, and Thierry Cruanes. 2006. Cost-Based Query Transformation in Oracle. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 1026–1036.

[6] Anindita Basak, Krishna Venkataraman, Ryan Murphy, and Manpreet Singh. 2017. *Stream Analytics with Microsoft Azure: Real-time data processing for quick insights using Azure Stream Analytics.* Packt Publishing Ltd.

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[8] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2 ed.). Prentice Hall Press, USA.

[9] Ashish Gupta, Venky Harinarayan, and Dallan Quass. 1995. Aggregate-Query Processing in Data Warehousing Environments. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 358–369.

[10] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 1–34.

[11] Dung Nguyen, Andre Luckow, Edward Duffy, Ken Kennedy, and Amy Apon. 2018. Evaluation of highly available cloud streaming systems for performance and price. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 360–363.

[12] Saeed Shahrivari. 2014. Beyond batch processing: towards real-time and streaming big data. *Computers* 3, 4 (2014), 117–129.

[13] Stratis Viglas and Jeffrey Naughton. 2002. Rate-Based Query Optimization for Streaming Information Sources. 37–48. https://doi.org/10.1145/564691.564697

[14] Wentao Wu, Philip Bernstein, Alex Raizman, and Christina Pavlopoulou. 2020. Cost-based Query Rewriting Techniques for Optimizing Aggregates Over Correlated Windows.

[15] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[16] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2019. The nebulastream platform: Data and application management for the internet of things. *arXiv preprint arXiv:1910.07867* (2019).